The International Journal of Parallel, Emergent and Distributed Systems
Vol. 00, No. 00, September 2015, 1–20

RESEARCH ARTICLE

GPU-OSDDA: A Bit-Vector GPU-based Deadlock Detection Algorithm for Single-Unit Resource Systems

Stephen Abell[a], Nhan Do[b], and John Jaehwan Lee[b*]

[a]*Boeing, Washington, USA*; [b]*Department of Electrical and Computer Engineering, Indiana University Purdue University Indianapolis, Indiana, USA*

(*v0.5 released September 2015*)

This article presents a GPU-based single-unit deadlock detection methodology and its algorithm, GPU-OSDDA. Our GPU-based design utilizes parallel hardware of GPU to perform computations and thus is able to overcome the major limitation of prior hardware-based approaches by having the capability of handling thousands of processes and resources, whilst achieving real-world run-times. By utilizing a bit-vector technique for storing algorithm matrices and designing novel, efficient algorithmic methods, we not only reduce memory usage dramatically but also achieve two orders of magnitude speedup over CPU equivalents. Additionally, GPU-OSDDA acts as an interactive service to the CPU, because all of the aforementioned computations and matrix management techniques take place on the GPU, requiring minimal interaction with the CPU. GPU-OSDDA is implemented on three GPU cards: Tesla C2050, Tesla K20c, and Titan X. Our design shows overall speedups of 6-595X over CPU equivalents.

Keywords: deadlock detection, resource allocation graph (RAG), GPU, CUDA, bit vector

1. Introduction

Modern systems are becoming increasingly complex, with hundreds or thousands of concurrent processes and resources being utilized and sharing resources at any given instant. This increased level of complexity has led to the higher possibility of systems entering a deadlock state. A deadlock in our context is a situation in which two or more competing processes are each waiting for the other to finish, and thus neither ever does [1].

In the past, many software-based deadlock detection algorithms [2–6] were written, but they lacked the speed necessary to make them viable in real world systems. As a result, researchers have developed hardware-based algorithms [7–11] that expanded upon the findings of these software algorithms. Hardware algorithms led to very fast and deterministic results but lacked the ability to handle an increasing amount of processes and resources (due to size constraints and hardware complexity), as would be seen in a real world system. In lieu of these findings though, we hypothesized that by adopting the methodologies found in the hardware approaches and exploiting their parallel nature on GPU, we may be able to devise a practical solution to the deadlock detection problem in systems with a large number of processes and resources (e.g., 4096 processes and 4096 resources). By

*Corresponding author. Email: johnlee@iupui.edu

ISSN: 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/17445760.YYYY.CATSid
http://www.informaworld.com

applying the hardware algorithmic methodologies and performing creative GPU optimizations, we have been able to provide a deadlock detection approach applicable to real world systems (e.g., computation time of a *less than a milisecond* in a 4096 process $\times$4096 resource system).

For this reason, we propose GPU-OSDDA, a GPU-based single-unit deadlock detection algorithm, which keeps track of all resource allocation events on the GPU. It is the CPU's responsibility to pass resource allocation event information to the GPU for computation. In this way, GPU-OSDDA serves as an interactive service to the CPU. When we mention the words *interactive service*, we refer to the limited interaction that the CPU is required to have with our algorithm. GPU-OSDDA is meant to run in the background, receiving resource event information from the Operating System (OS) and then provide notification to the CPU in the event of any deadlock occurring in the system. In this way, our algorithm provides an unobtrusive notification to the CPU (or OS) regarding the state of its resource events.

A summary of our contribution is as follows: (1) Proposing novel algorithmic methods for GPU acceleration of deadlock detection, achieving two orders of magnitude speedup; (2) Utilizing a bit-vector technique for storing algorithm matrices, thus reducing memory usage dramatically; (3) Handling thousands of processes and resources, while achieving real-world run-times; (4) Offering as an interactive service to the OS, requiring minimal interaction with the CPU; (5) Bridging the gap between problem size and run-time of deadlock detection algorithms.

2. Background

2.1 Related Work

There have been a multitude of software-based deadlock detection algorithms proposed in the past that handle resource events in single-unit resource systems. In 1972, Holt [2] first introduced a resource allocation graph-based deadlock detection approach that had an $\mathcal{O}(m \times n)$ run-time complexity, where m and n are the process and resource amounts, respectively. Following this development, Leibfried [6] designed an algorithm that utilized the adjacency matrix. Leibfried's approach used matrix multiplication in order to determine reachability information, which led to an algorithm with an $\mathcal{O}(m^3)$ run-time complexity. Later, Kim and Koh [4] devised a tree-based algorithm that improved upon the prior deadlock detection run-time. Their tree-based algorithm was able to detect deadlock in $\mathcal{O}(1)$ run-time; however, the caveat to this approach was that it required an $\mathcal{O}(m+n)$ run-time for the resource release phase of the algorithm. The completion of this release phase was required for the algorithm to handle the next invocation of deadlock detection.

In recent years, there has been a progression towards parallel hardware-based algorithms to detect deadlock. These algorithms are deterministic and have accomplished low run-time complexities in hardware. One of such algorithms, known as HDDU, was developed by Xiao and Lee [8] in 2007. This algorithm had a deadlock detection run-time of $\mathcal{O}(1)$ and a detection preparation run-time of $\mathcal{O}(min(m,n))$. Later, Xiao and Lee developed a new approach to classifying resource events in a single-unit resource system. This development led to a new algorithm known as $\mathcal{O}(1)$ Single-Unit Deadlock Detection Algorithm (OSDDA) [9]. By utilizing the new classification of resource events, deadlock preparation was able to be completed in $\mathcal{O}(1)$ time. As a result, OSDDA was able to achieve an overall run-time complexity of $\mathcal{O}(1)$ in hardware. The basis of our algorithm is rooted in the methodology of OSDDA [9]. The core operation of OSDDA is based upon a classification of re-

source events in the system which is further discussed in Section 2.4. Note that the problem of deadlock detection for multi-unit resources is out of scope of this article.

2.2 The RAG and its Adjacency Matrix Representation

The events occurring between processes and resources of a system are represented as a bipartite graph known as the Resource Allocation Graph (RAG). It contains two disjoint sets: a process set P and a resource set Q. There are two types of edges between these disjoint sets. The first edge type is known as a *resource request* edge. It is a directed edge from a process node p_i in set P to a resource node q_j in set Q that denotes process p_i has requested resource q_j. The second edge type is known as a *resource grant* edge. This edge is a directed edge running from a resource node q_j in Q to a process node p_i in P that denotes resource q_j has been granted to process p_i. The RAG can also be represented as two separate adjacency matrices: *Adjacency Request* (AR) and *Adjacency Grant* (AG), which hold the resource request and grant information, respectively [8]. AG[] and AR[] can be defined as follows:

$$AG[j][i] = \begin{cases} 1 \text{ if } \exists q_j \rightarrow p_i, \\ 0 \text{ otherwise.} \end{cases} \qquad AR[i][j] = \begin{cases} 1 \text{ if } \exists p_i \rightarrow q_j, \\ 0 \text{ otherwise.} \end{cases}$$

where $1 \leq i \leq m$ and $1 \leq j \leq n$.

2.3 Terms and Assumptions

For the sake of understanding the computations of deadlock detection in GPU-OSDDA, we here introduce relevant terms.

Definition 1. An interactive service is an application or program that runs in the background with limited interaction with the operating system.

Definition 2. A single-unit resource is a resource that serves at most one process at any given instant.

Example single-unit resources can be as simple as a USB port, a printer port, or a network port, or as complex as files, memory pages, or every computing node in a cluster of machines or cloud.

Definition 3. A single-unit request system is a system in which a process may request only one unit at a time and thus has at most one outstanding request [12].

Definition 4. A system is in an expedient state if any request for an available unit is granted immediately [2].

The system under consideration is a single-unit resource system with m number of processes and n number of resources, which we refer to as an $m \times n$ system. It is also a single-unit request and expedient system. In such a system, a single-unit request also means requesting a single resource only per request (i.e., a request for multiple resources by one command is not valid).

Definition 5. A sink process node is a non-blocked process node (no outgoing edge) with at least one granted resource (incoming edge) [2] .

Definition 6. An active process is a process which has no pending resource request (no outgoing edge) but may have granted resources (incoming edges).

Definition 7. A node v_i is reachable from a node v_j if and only if there exists a path that starts from v_j and ends at v_i [2]. Thus, v_i is called a reachable node of v_j.

Figure 1 illustrates a Resource Allocation Graph (RAG) containing three processes and three resources. Solid arrows denote resource grant edges, and dotted arrows denote request edges. Black dots mean resource units. In the figure, p_0 is an active process and its node is a sink node, whereas p_1 and p_2 are blocked waiting for q_2. At the moment, p_0 is a reachable sink node of all resources.

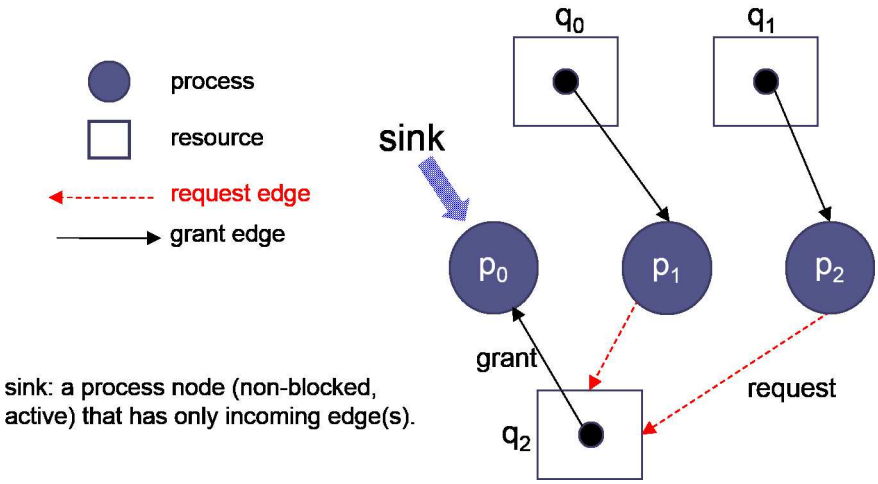


Figure 1. A 3×3 single-unit resource single-unit request expedient system.

The proposed algorithm adheres to the following assumptions:

(1) Each resource contains a single unit (see Definition 2). Thus, a cycle in the RAG is a necessary and sufficient condition for deadlock [9].
(2) A process requests one resource at a time (see Definition 3). Thus, a process is blocked as soon as it requests an unavailable resource [2].
(3) A resource is granted to a process immediately if the resource is available. As a result, the entire system is always in an *expedient* state (see Definition 4) [2].
(4) Resource events are managed centrally (e.g., by the OS).

These assumptions are very typical ones made in deadlock research. Note that the concerns of livelock, priority inversion, etc. are out of scope of this article.

2.4 Underlying Theory

OSDDA is truly unique because its overall algorithm run-time is $\mathcal{O}(1)$ in hardware. It is able to achieve this by performing parallel computations on a RAG based on the classification of resource events in the system. The three types of resource events in OSDDA are: *granted resource requests*, *blocked resource requests*, and *resource release* [9]. The resource events and deadlock detection capability of OSDDA are briefly discussed in Sections 2.4.1 and 2.4.2.

2.4.1 Resource Events

Let us first discuss the *resource request granted* event. For this event to occur, when a process p_i requests a resource q_j, q_j has to be available (not granted to another process) and p_i needs to be an active process. This means that resource q_j must not have any incoming or outgoing edge and process p_i may have incoming edges but no outgoing edge (since the system is in an expedient state (see Definition 4)). If these criteria are satisfied, resource q_j may be granted to process p_i. This event causes q_j to change its reachable sink node.

Next we'll discuss the *resource request blocked* event. When a process p_i requests a resource q_j and there is no available unit of q_j, the process p_i is blocked. Prior to the request, process p_i has to be an *active process*, and thus, p_i has no outgoing edge when the request is made. By definition of an *active process*, p_i could have already been granted resources and as a result have incoming edges. Furthermore, resource q_j has an outgoing edge (as it is not available) and may have incoming edges (pending requests). As a result, two scenarios of the *resource request blocked* event exist:

(1) Block (i) - Before the request is blocked, p_i has no incoming edges; q_j has one outgoing edge; q_j may or may not have incoming edges. After the request is blocked, only a request edge $p_i \rightarrow q_j$ is inserted in the RAG [9].
(2) Block (ii) - Before the request is blocked, p_i has incoming edges; q_j has one outgoing edge; q_j may or may not have incoming edges. After the request is blocked, for all resources belonging to p_i's sub-tree, their sink nodes are set to q_j's sink node and their reachable processes and resources are also updated to include q_j's reachable nodes [9].

Finally, we'll discuss the *resource release* event. For process p_i to release its resource, it must be an *active process* by having no outgoing edge. While servicing the *resource release* event, the algorithm must determine if resource q_j has any pending resource requests (incoming edges). If q_j has a pending request from a process p_t, q_j is granted to p_t after the release due to the system being in an expedient state. Depending on if resource q_j has pending requests, two separate *resource release* scenarios exist.

(1) Release (i) - Before released, q_j has no incoming edges; p_i may have one or more incoming edges. After the resource is released, only a grant edge $q_j \rightarrow p_i$ is removed in the RAG, and thus, q_j is no longer reachable to p_i [9].
(2) Release (ii) - Before released, q_j may have one or more incoming edges; p_i may have one or more incoming edges; p_t may or may not have incoming edges. After release, q_j is assigned to p_t. In this case, the sink nodes of all of q_j's sub-tree resources are changed to p_t, and also they are no longer reachable to p_i [9].

Figure 2(a) shows an example RAG of a 3×3 system consisting of three processes (p_0, p_1, p_2) and three resources (q_0, q_1, q_2). Since this is a single-unit system, each resource has one unit. Accompanying the RAG in Figure 1 are the associated adjacency matrices AG and AR that are formed via the prior definitions. In the RAG, there exist three *resource grant* edges ($q_0 \rightarrow p_0$, $q_1 \rightarrow p_2$, and $q_2 \rightarrow p_0$) and two *resource request* edges ($p_1 \rightarrow q_0$ and $p_2 \rightarrow q_2$). On each resource event, the operating system sends the event information to the GPU so that it may update its RAG (i.e., AG and AR) and initiate deadlock detection if necessary.

Furthermore, by looking at Figure 2(b), it can be seen that under the resource release event where p_0 releases q_0, q_0 is then granted to the blocked process p_1.

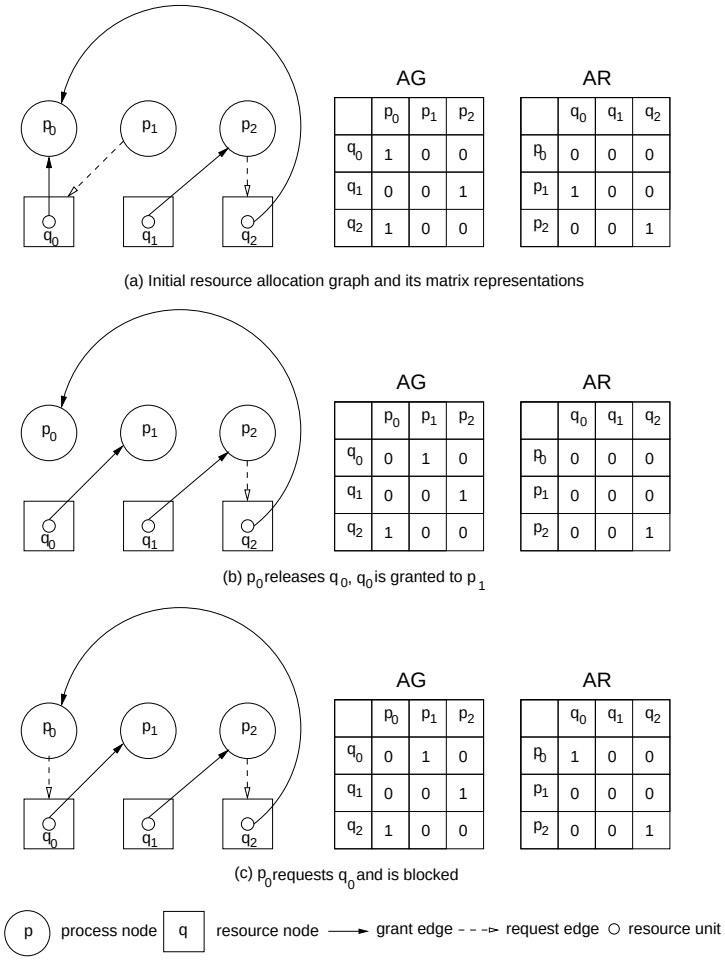


Figure 2. A 3×3 RAG incurring resource events.

This is an example of the system being in an *expedient* state. Notice also that AG and AR have been updated accordingly. Lastly, in Figure 2(c), process p_0 requests q_0 and is blocked due to q_0 having been granted to process p_1.

2.4.2 O(1) Deadlock Detection

It is known that as long as the *sink process node* for every resource in the system has been identified, then deadlock can be detected in $\mathcal{O}(1)$ time as reported in [4] and [8]. We know the reachable *sink process node* of a resource q_j is process p_i if and only if p_i is a *sink process node* and a path from resource q_j to process p_i exists. A cycle occurs in the system when the *sink process node* (say p_i) of a resource (say q_j) requests the resource. By our system assumptions, a cycle in the RAG is a necessary and sufficient condition for deadlock, and thus, under this scenario, a deadlock exists.

To achieve $\mathcal{O}(1)$ run-time of deadlock detection, OSDDA maintains the sink information for all resources in the system for use in upcoming invocations of deadlock detection. The sink information is stored in a matrix known as Sink.

Furthermore, for a release (ii) event, the OSDDA algorithm needs to identify resources on the sub-tree of the released resource (q_j) as well as those on the sub-tree of the process acquiring q_j [8]. For this, OSDDA utilizes the ReachableResource or RR and the ReachableProcess or RP matrices to maintain information on what resources and processes are reachable from every resource, respectively [8]. The matrices are defined as follows:

$$Sink[j][i]_{n\times m} = \begin{cases} 1 \text{ if } p_i \text{ is } q_j\text{'s reachable sink node,} \\ 0 \text{ otherwise.} \end{cases}$$

$$RR[j][k]_{n\times n} = \begin{cases} 1 \text{ if a path exists from resource } q_j \text{ to } q_k \\ \quad \text{or } k = j, \\ 0 \text{ otherwise.} \end{cases}$$

$$RP[j][i]_{n\times m} = \begin{cases} 1 \text{ if a path exists from resource } q_j \text{ to } p_i, \\ 0 \text{ otherwise.} \end{cases}$$

3. Our Methodology

3.1 Introduction

During the development process of our preliminary version of GPU-OSDDA, we implemented two versions, one with characters and the other with integers to represent our matrix elements. For both the character and integer-based approaches, time was spent optimizing and tweaking GPU code to ensure that occupancy was high, coalesced memory accesses were occurring, and threads were kept busy, following GPU programming guidelines. This enabled us to maintain a high IPC ratio and maximize the memory bandwidth for our problem. We utilized the NVIDIA Compute Visual Profiler to gauge our results at each step and came to a point where we were satisfied with the optimizations. However, with all optimizations complete, we were only able to achieve 3-24X speedup over our CPU implementation, dubbed CPU-OSDDA. These speedups, while an improvement, did not grant us the kinds of speedups we were looking for.

Using our initial approach as a baseline for measuring performance, we began rework on the algorithm. Our new approach took a drastically different approach on matrix storage and algorithm computation. We thought this different approach was necessary in order to maximize speedup and yield an algorithm that would be applicable to real world systems. Thus, we decided to implement our entire algorithm with integer length bit-vectors. We hypothesized that this approach would reduce our memory footprint by a factor of 32, thus allowing for an increasing amount of processes and resources the algorithm could handle, as well as simplify and accelerate the bit-wise computations of our algorithm. Advantages of this approach are discussed throughout the remaining subsections, as well as how GPU-OSDDA handles each resource event type. This kind of GPU-targeted bit-packed approach has never been well reported in the literature as far as we know.

3.2 Novel Bit-Vector Design

Since GPU-OSDDA is based on a single-unit system, all values that indicate the state of a process or resource in the system can be represented as binary values (0,1). In this case, instead of using an 8 or 32-bit variable to hold a 1-bit value, we bit-pack 32 processes or resources into a single 32-bit unsigned integer. Figure 3 shows how we would create a 128×128 adjacency matrix using 32-bit unsigned integers, where each box in the image represents a 32-bit unsigned integer. Similarly, we could create an adjacency matrix where the rows/columns are reversed.

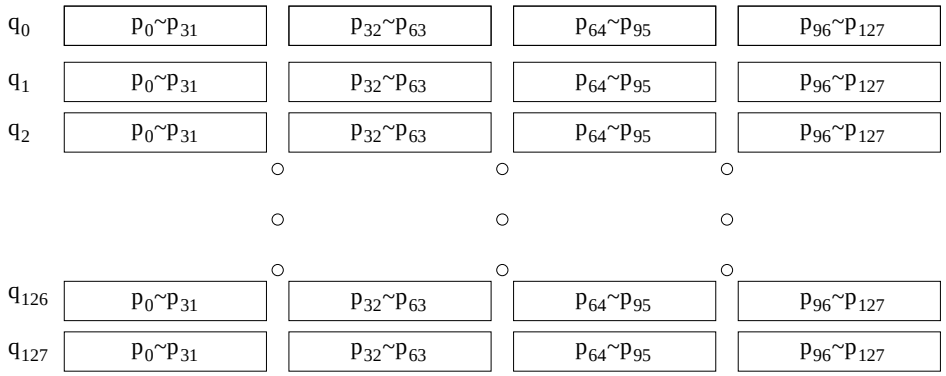


Figure 3. A 128×128 Bit-vector adjacency matrix.

Additionally, Table 1 describes variables used throughout the remaining algorithm descriptions using Figure 3. By using left bit-shifting to achieve multiplication and right bit-shifting to achieve division in our algorithm, the LIPR and LINTBITS values are necessary to gain proper offsets when calculating matrix indices. This is due to the fact that each bit-shift (left or right) implies a change in magnitude by a power of two.

Table 1. Common algorithm variables for GPU-OSDDA

Variable	Description	Values in Figure 3
INTS_PER_ROW (IPR)	The number of integers in a bit-packed row	4
INT_BITS	The number of bits per unsigned integer	32
LIPR	Equivalent to log_2(INTS_PER_ROW)	2
LINTBITS	Equivalent to log_2(INT_BITS)	5

Now that all algorithm critical information has been discussed, we present the overall kernel structure with the pseudo-code in Algorithm 1. The pseudo-code presents the kernels called upon each resource event type. In lines 4-5, GPU-OSDDA handles the *resource request granted* event. The *Request_Granted* kernel launches a single block containing a single thread to perform the updates discussed in Section 3.3.

Algorithm 1 Overall Kernel Structure

```
1:  // Refer to Table 1 for variable definitions
2:  // Where TILE_DIM equals M or N ÷ INT_BITS (32 bits per unsigned int)
3:  // Input: event info (p_i, q_j, p_t, event_type), Output: deadlock or not
4:  if Resource Request Granted event then
5:      Request_Granted<<<1,1>>>(event, AG, Sink, RP)
6:  else if Resource Request Blocked event then
7:      DeadlockCheck_Init<<<1,1>>>(event, Sink, AR, deadlock)
8:      if deadlock == false then
9:          BitMatrix_Transpose<<<TILE_DIM,TILE_DIM>>>(AG_temp, AG)
10:         Tile_Transpose<<<TILE_DIM,TILE_DIM>>>(AG_trans, AG_temp)
11:         Row_Reduction<<<1,IPR/2>>>(AG_trans[event→ p_i * IPR], phold)
12:         if phold == true then
13:             Request_Blocked<<<N,IPR>>>(event, Sink_temp, Sink, RR, RP)
14:         end if
15:     else
16:         Handle Deadlock
17:     end if
18: else if Resource Released event then
19:     Release_Resource<<<1,1>>>(event, AG)
20:     BitMatrix_Transpose<<<TILE_DIM,TILE_DIM>>>(AR_temp, AR)
21:     Tile_Transpose<<<TILE_DIM,TILE_DIM>>>(AR_trans, AR_temp)
22:     Row_Reduction<<<1,IPR/2>>>(AR_trans[event→ q_j * IPR], pwait)
23:     if pwait == 0 then
24:         Update_Sink_RP<<<1,1>>>(event, Sink, RP)
25:     else
26:         Update_AG_AR<<<1,1>>>(event, AG, AR)
27:         Release_Update_Reachability<<<N,IPR>>>(event, Sink, RP, RR)
28:     end if
29: else
30:     Not a valid event
31: end if
```

Lines 6-17 in Algorithm 1 handle the *resource request blocked* event. The *DeadlockCheck_Init* kernel in line 7 launches a single block containing a single thread. Utilizing the Sink[] matrix previously discussed, the kernel checks for deadlock and updates its status accordingly. Line 8 checks the system's deadlock status. If deadlock exists, the algorithm notifies the CPU of the deadlock status. Otherwise, lines 8-9 perform a bit-wise matrix transpose on the AG[] matrix, which allows for coalesced memory accesses in the *Row_Reduction* kernel in line 11. The reduction kernel launches a single block with a number of threads equal to the number of integers per matrix row (INTS_PER_ROW or IPR) divided by two. The reduction kernel determines if the blocked process holds any additional resources. If the process does not hold additional resources, sink nodes do not change (block (i) event) and no additional computation is needed. If the process does hold additional resources, we launch the *Request_Blocked* kernel in line 13 to update Sink[], RR[], and RP[]. This kernel launches N blocks with INTS_PER_ROW threads per block to facilitate parallel computation. The *resource request blocked* functionality is discussed in detail in Section 3.4, while the *BitMatrix_Transpose*, *Tile_Transpose*, and *Row_Reduction* kernels are discussed in Section 3.6.

Lastly, lines 18-28 handle the *resource release* event. In line 19, the *Resource_Release* kernel launches a single block with a single thread to perform the update to AG[] reflecting the resource release. Lines 20-21 perform a similar function to what was done in lines 9-10, except this time we transpose and check the AR[] matrix. The *Row_Reduction* kernel tells us if any processes are waiting for the released resource. If no processes are waiting, we launch the *Update_Sink_RP* kernel in line 24, which performs the updates on the Sink[] and RP[] matrices. Otherwise, we grant the released resource to a new process with the *Update_AG_AR* kernel in line 26. Following the resource grant, we update the reachability information in the system by calling the *Release_Update_Reachability* kernel in line 27. The *release resource* event is discussed in detail in Section 3.5.

3.3 Handling a Resource Request Granted Event

To handle a resource request granted event, GPU-OSDDA launches a kernel with a single block containing a single thread. The computation involved in this event does not advocate parallelism; however, GPU-OSDDA manages and maintains the RAG on the GPU, which makes it necessary to launch this small kernel. Algorithm 2 shows the assignments made in our kernel. Since we utilize a bit-packing technique to represent all algorithm matrices, we have to use a special method of referencing the correct process/resource pair for assignment.

In order to find the correct bit corresponding to the process in the grant event, handled by the Request_Granted computation, we perform in line 3 the modulo of p_i by INT_BITS. After obtaining the correct bit in the integer, we find the exact integer index to be altered in each adjacency matrix. This is computed in line 4 by left-shifting the row we want (q_j) by log_2(INTS_PER_ROW). To this we add the process number right-shifted by log_2(INT_BITS). The sum of these two numbers yields the index of the integer we want to alter in the adjacency matrix. This computation is very similar to calculating the global thread ID of one dimensional multiblock grid of a GPU programming model. Bit-shifting (left and right) are used instead of multiplication and division, respectively, for efficiency. Here the size of the matrices should be powers of two, so that we can perform the bit-shift operations for index calculations. Note that M and N do not need to be equal in size, but they must be powers of two. If M and N are not equal, the INTS_PER_ROW value will change depending on which matrix we address. For ease of explanation in this article, we assume that M and N are equal. Note however that for those cases where the number of processes or the number of resources is not power of two, a common zero-padding method can be used to make the matrix size power of two. Nonetheless, as information is bit-packed, it will incur neither much space overhead nor much computation work as padded values are all zeros.

In order to perform assignments to the adjacency matrices, we perform bit-wise OR computations with the appropriate mask. The mask is created by shifting a 1 into the location specified by the bit variable we calculated in line 3. Upon performing the bit-wise OR operations in lines 8-10, our Request_Granted kernel is complete.

Algorithm 2 Request_Granted⋘1,1⋙

```
1:  // Refer to Table 1 for variable definitions
2:  // Determine index variables - integer index and bit to alter
3:  bit ← p_i mod INT_BITS
4:  idx ← q_j ≪ LIPR + p_i ≫ LINTBITS
5:
6:  // Update AG[], Sink[], and RP[] to reflect resource grant
7:  // All assignments are bit-wise computations using index variables
8:  AG[idx] |= (1 ≪ (INT_BITS − (bit + 1)))
9:  Sink[idx] |= (1 ≪ (INT_BITS − (bit + 1)))
10: RP[idx] |= (1 ≪ (INT_BITS − (bit + 1)))
```

As a summary, in the Request_Granted kernel, it can be seen that the AG[] matrix is updated to reflect the assignment $q_j \rightarrow p_i$. Similarly, by resource q_j being granted to p_i, p_i becomes the new reachable sink node of resource q_j, denoted by the Sink[] matrix. It follows that process p_i is reachable from resource q_j as denoted by the RP[] matrix assignment. Figure 4 summarizes the actions taken by the resource request granted kernel.

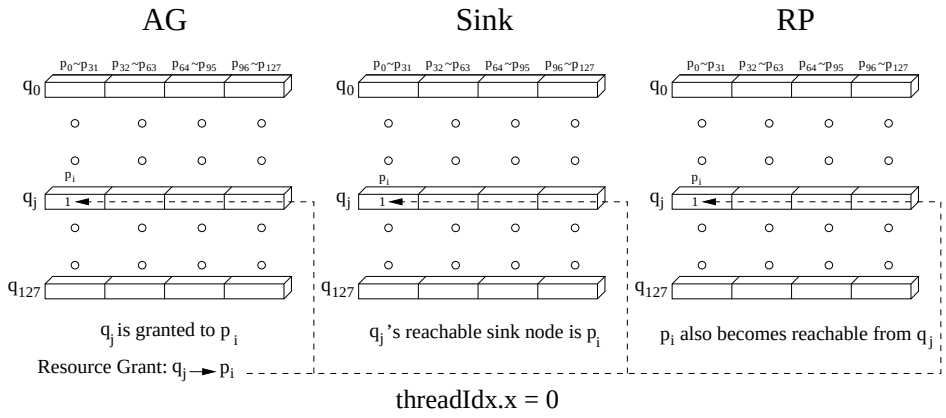


Figure 4. Resource request granted.

3.4 Handling a Resource Request Blocked Event

GPU-OSDDA handles a resource request blocked event through several stages. The initial step, which we denote as DeadlockCheck_Init (Algorithm 3), checks whether or not the requesting process is the current sink node of the requested resource. According to [9], if a resource request event occurs where the requesting process is the current sink node of the resource being requested, a cycle forms in the RAG and a deadlock occurs. Otherwise, AR[] is updated to reflect the blocked request of $p_i \rightarrow q_j$.

The DeadlockCheck_Init kernel utilizes a similar technique seen in the Request_Granted kernel to determine the indices needed for its computations. First, we determine which bit needs to be checked and/or set in the kernel. We first determine the bit to be checked in the Sink[] matrix in line 5. Since we want to check a single bit in an integer, we perform the modulo of process p_i by INT_BITS. Similarly, we need a bit for the AR[] matrix. The reason for building two separate indices is that the Sink[] and AR[] matrices take the form of *resource* $\times$ *process* and *process* $\times$ *resource*, respectively. For the AR[] matrix, we gain the bit to check by performing the modulo of resource q_j by INT_BITS in line 6. As can be seen in Algorithm 3, we continue by constructing two separate global indices; *sidx* and *aidx* in lines 7-8. The *sidx* index yields the position of the integer we want to check in the Sink[] matrix, while the *aidx* index yields the position of the integer for assignment in the AR[] matrix. The combination of both the global integer index and the associated bit index enables us to check or alter a single bit in the appropriate adjacency matrix.

If a deadlock occurs (checked in line 12), then we update the deadlock detection flag in line 13 for the CPU to handle the deadlock event. Otherwise, the resource request is blocked in line 15 by updating the value corresponding to the request $p_i \rightarrow q_j$ in the AR[] matrix.

Algorithm 3 DeadlockCheck_Init⋘1,1⋙

```
1:  // Refer to Table 1 for variable definitions
2:  // Determine index variables - integer index and bit to alter
3:  // We have two sets of indices as Sink[] is resource ×process
4:  // and AR[] is process × resource
5:  sbit ← p_i mod INT_BITS
6:  abit ← q_j mod INT_BITS
7:  sidx ← q_j ≪ LIPR + p_i ≫ LINTBITS
8:  aidx ← p_i ≪ LIPR + q_j ≫ LINTBITS
9:
10: // If current requested resource's sink node is the requesting process
11: // then a deadlock exists. Otherwise, block the request by updating AR[].
12: if (Sink[sidx] & (1 ≪ (INT_BITS − (sbit + 1))) == 1) then
13:     Update Deadlock Flag
14: else
15:     AR[aidx] |= (1 ≪ (INT_BITS − (abit + 1)))
16: end if
```

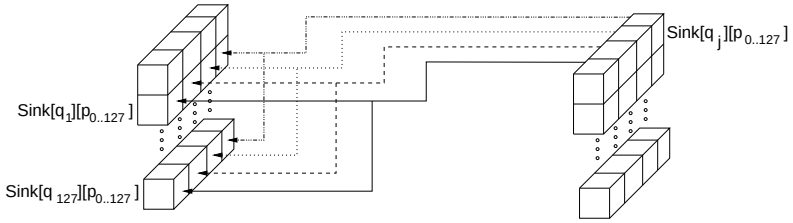
If a deadlock does not occur, reachability information of the RAG needs to be updated if the requesting process holds additional resources. Otherwise, the reachable sink nodes do not change, so no additional computation is necessary. The task of updating reachability information for a RAG is computationally expensive, unlike $\mathcal{O}(1)$ of OSDDA [9]. Nevertheless, our implementation of the reachability update computation benefits greatly from the parallelism offered by the GPU and is further accelerated by our bit-vector approach to the algorithm. Algorithm 4 shows the pseudo-code for our Request_Blocked kernel, which performs the reachability update. As can be seen by the kernel overview in Algorithm 1, we launch N blocks with INTS_PER_ROW threads per block. This kernel structure allows us to perform all bit-wise computations in this kernel simultaneously, except for the serialization of the Sink[], RR[], and RP[] updates per thread. The bit-vector approach we implement allows us to perform computations for 32 processes or resources per integer index in an adjacency matrix. This approach granted us a dramatic speedup in the run-time of our algorithm, which will be depicted in the Experimentation and Results section of this article. The first step in our Request_Blocked kernel (Algorithm 4) is to determine the indices in our adjacency matrices (lines 3-7).

We also allocate a temporary sink matrix, Sink_temp[], on the GPU which takes on the values of the Sink[] matrix. The Sink_temp[] matrix is used to check values of the Sink[] matrix without the risk of race conditions between the read and write cycles of the Sink[] matrix. After performing our check in line 12, the Sink[], RR[], and RP[] matrices are updated according to [9] in lines 13-15. Line 13 makes the sink node of all resources on the subtree of p_i equal to q_j's sink node. Then in lines 14-15, the resources on p_i's subtree include the reachable resources and processes of q_j. Following Algorithm 4, a summary of the operations performed for the Request_Blocked kernel is provided.

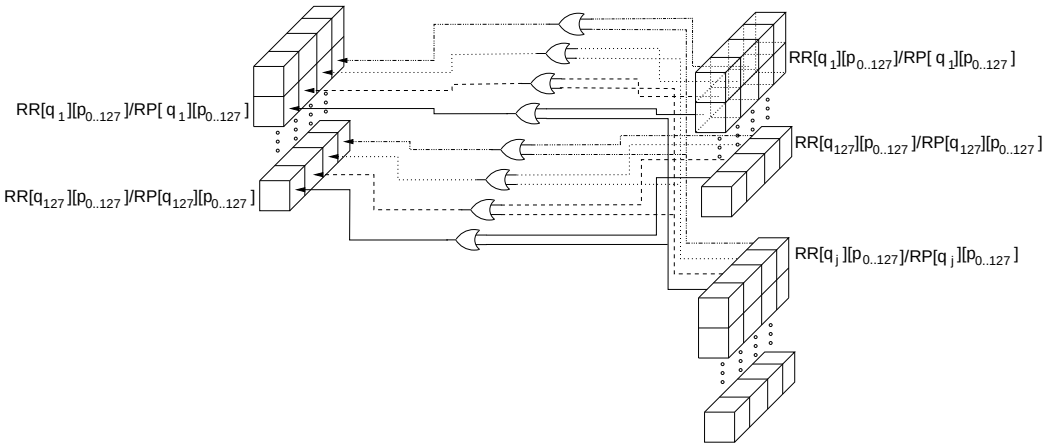
Algorithm 4 Request_Blocked⋘N,IPR⋙

```
1:  // Refer to Table 1 for variable definitions
2:  // Indexing variables for reachability update computation
3:  row ← blockIdx.x
4:  col ← p_i ≫ LINTBITS
5:  bit ← p_i mod INT_BITS
6:  tid ← threadIdx.x
7:  idx ← row ≪ LIPR + col
8:
9:  // For all the resources that belong to the subtree of pi,
10: // their sink nodes are now set to q_j 's; their reachable
11: // resource and process nodes include q_j 's.
12: if ((Sink_temp[idx] & (1 ≪ (INT_BITS − (bit + 1)))) == 1) then
13:     Sink[row × IPR + tid] ← Sink[q_j × IPR + tid]
14:     RR[row × IPR + tid] ← RR[row × IPR + tid] | RR[q_j × IPR + tid]
15:     RP[row × IPR + tid] ← RP[row × IPR + tid] | RP[q_j × IPR + tid]
16: end if
```

As in [9], for all resources belonging to p_i's sub-tree, their sink nodes are set to q_j's sink node. Their RR[] and RP[] matrices are also updated to include q_j's reachable nodes. The biggest advantage we gain during this computation is that per each assignment or bit-wise OR operation, we effectively update 32 process/resource pairs per thread. More specifically, each row of a matrix is handled per block with each column (integer) being handled by a thread. Figure 5 depicts the operations taking place during the reachability computation, where resources q_1 and q_{127} are updated to have a sink node of p_i, and q_j is assumed to be q_0 (i.e., the first row). Figure 5(a) depicts line 13 in the Request_Blocked kernel where the reachable sink nodes are updated. Figure 5(b) depicts lines 14-15 in the Request_Blocked kernel, where the logical OR gates are performing the task of updating the reachable processes and resources of p_i's subtree resources to include q_j's reachable processes and resources. In addition, Figure 5(b) shows the logic diagram for a single matrix, but note that an identical operation is occurring for both matrix RR[] and RP[].



(a) Update reachable sink node (line 13)



(b) Update reachable processes and resources (lines 14−15)

Figure 5. Updating sink nodes and reachability for Request_Blocked events.

3.5 Handling a Resource Release Event

In handling a *resource release* event, GPU-OSDDA first has process p_i release resource q_j by updating AG[], as the pseudo-code in Algorithm 5 depicts. We first determine the bit that represents the process p_i releasing resource q_j in line 3. Following a familiar procedure, we compute p_i modulo INT_BITS to represent our process bit. Then to determine the integer index into the AG[] matrix that we need to alter, we compute q_j left-shifted by log_2(INTS_PER_ROW) to give us the proper row of AG[] in line 4. We then add this to p_i right-shifted by log_2(INT_BITS) to give us the column integer that we want to address. This sum then yields the index to address in AG[]. From there, the operation in line 7 performs a bit-wise AND operation to update AG[] reflecting that process p_i released resource q_j.

Algorithm 5 Release_Resource⋘1,1⋙

```
1: // Refer to Table 1 for variable definitions
2: // Determine index variables - integer index and bit to alter
3: bit ← p_i mod INT_BITS
4: idx ← q_j ≪ LIPR + p_i ≫ LINTBITS
5:
6: // Release resource from AG matrix by clearing the bit
7: AG[idx] &= ∼ (1 ≪ (INT_BITS − (bit + 1)))
```

GPU-OSDDA then checks if a process is waiting on q_j by performing a reduction on AR_trans$[q_j][]$ (transpose of the AR[] matrix). In the event that this reduction returns 0, it informs us that no process is waiting on q_j and that it belongs to a release event (i), explained in Section 2.4.1 and detailed in [9]. From there, GPU-OSDDA updates the Sink[] and RP[] matrices to indicate that q_j has no sink and that p_i is no longer reachable from q_j. Algorithm 6 depicts the update process of Sink[] and RP[]. Since both Sink[] and RP[] are *resource* × *process* matrices, we are able to utilize the same bit and index to update necessary information. Notice the procedure in lines 3-4 in Algorithm 6 is similar in terms of finding the appropriate bit and index. After computing this information, we perform a bit-wise AND operation in lines 7 and 9 to clear the corresponding bit in the adjacency matrices.

Algorithm 6 Update_Sink_RP⋘1,1⋙

```
1: // Refer to Table 1 for variable definitions
2: // Determine index variables - integer index and bit to alter
3: bit ← p_i mod INT_BITS
4: idx ← q_j ≪ LIPR + p_i ≫ LINTBITS
5:
6: // q_j is isolated; thus q_j has no sink - clear bit
7: Sink[idx] &= ∼ (1 ≪ (INT_BITS − (bit + 1)))
8: // p_i is no longer reachable from q_j either - clear bit
9: RP[idx] &= ∼ (1 ≪ (INT_BITS − (bit + 1)))
```

If the reduction of AR$[][q_j]$ is not equal to 0, this indicates that the release event belongs to the release (ii) scenario, explained in Section 2.4.1 and detailed in [9]. In this case, GPU-OSDDA updates AG[] and AR[] to indicate that the released resource is granted to a waiting process. Algorithm 7 depicts the update process for AG[] and AR[]. Since the AG[] and AR[] matrices take the form of *resource* × *process* and *process* × *resource* respectively, they both need their own bit and global index variables to update the correct bit. As performed for all of our updates thus far, we find the correct bit by computing the modulo of the bit we want with INT_BITS in lines 5-6. Following that, lines 7-8 perform familiar computations to find the integer index into the adjacency matrix that we want to update. Finally, we update AG[] and AR[] by performing bit-wise OR operations on the corresponding index and bit in lines 11-12.

Algorithm 7 Update_AG_AR⋘1,1⋙

```
1: // Refer to Table 1 for variable definitions
2: // Determine index variables - integer index and bit to alter
3: // We have two sets of indices as AG[] is resource ×process
4: // and AR[] is process × resource
5: tbit ← p_t mod INT_BITS
6: qbit ← q_j mod INT_BITS
7: tidx ← p_t ≪ LIPR + q_j ≫ LINTBITS
8: qidx ← q_j ≪ LIPR + p_t ≫ LINTBITS
9:
10: // q_j is now granted to p_t - set appropriate bits
11: AG[qidx] |= (1 ≪ (INT_BITS − (qbit + 1)))
12: AR[tidx] |= (1 ≪ (INT_BITS − (tbit + 1)))
```

Following this step, reachability and sink information needs to be updated.

Algorithm 8 provides our pseudo-code in handling the update process. The Release_Update_Reachability kernel takes advantage of the parallelism provided by the GPU. For this kernel, we launch N blocks with a number of threads equal to INTS_PER_ROW for each block. The kernel also utilizes the same methods as prior kernels to check, set, and clear bits in our adjacency matrices.

Algorithm 8 Release_Update_Reachability⋘N,IPR⋙

```
1:  // Refer to Table 1 for variable definitions
2:  // Shared variable holds new sink information
3:  __shared__ newSink[IPR]
4:
5:  // Determine index variables - integer index and bit to alter
6:  // Multiple indices are needed since we reference three different
7:  // variables: p_t, p_i, and q_j
8:  row ← blockIdx.x
9:  tid ← threadIdx.x
10: tbit ← p_t mod INT_BITS; pbit ← p_i mod INT_BITS; qbit ← q_j mod INT_BITS
11: tcol ← p_t ≫ LINTBITS;   pcol ← p_i ≫ LINTBITS;   qcol ← q_j ≫ LINTBITS
12: tidx ← row ≪ LIPR + tcol; pidx ← row ≪ LIPR + pcol; qidx ← row ≪ LIPR + qcol
13:
14: // Initialize newSink so p_t is the new sink node and synchronize threads
15: newSink[tid] ← 0
16: if tid == tcol then
17:     newSink[tcol] |= (1 ≪ (INT_BITS − (tbit + 1)))
18: end if
19: __syncthreads()
20:
21: // For q_j and its sub-tree resources
22: if (RR[qidx] & (1 ≪ (INT_BITS − (qbit + 1))) == 1) then
23:     // Assign the new sink node information
24:     Sink[row × IPR + tid] ← newSink[tid]
25:
26:     // Avoids writing to the same location for each thread
27:     // although the remaining threads go inactive
28:     if tid == 0 then
29:         // p_i is no longer reachable
30:         RP[pidx] & = ∼ (1 ≪ (INT_BITS − (pbit + 1)))
31:
32:         // For p_t's sub-tree resources that were able to reach q_j
33:         if (RP[tidx] & (1 ≪ (INT_BITS − (tbit + 1))) == 1) then
34:             // q_j is no longer reachable
35:             RR[qidx] & = ∼ (1 ≪ (INT_BITS − (qbit + 1)))
36:         else
37:             // p_t becomes reachable
38:             RP[tidx] |= (1 ≪ (INT_BITS − (tbit + 1)))
39:         end if
40:     end if
41: end if
```

To start, in line 3 we create an array in shared memory called *newSink*[], which we use to update sink information later in the kernel. Lines 8-9 assign the block and thread variables to row and tid, respectively, which are used for calculating the bit, column, and index variables. It can be seen that the same familiar process to find needed bits (line 10), columns (line 11), and indices (line 12) has been performed. This kernel, however, has every block handle a row in the matrices involved in computation. In lines 15-19, the newSink shared variable is populated to hold the new sink node, i.e., process p_t. In line 22, we check the RR[] matrix to obtain all of q_j's subtree resources. After finding all of q_j's subtree resources, we assign them the new sink node of p_t in line 24. Since p_i released resource q_j, p_i is no longer reachable from q_j and is removed from the RP[] matrix in line 30. In line 33, we check if p_t's subtree resources were able to reach q_j. If yes, q_j is no longer reachable from those resources so we remove q_j from the RR[] matrix in line 35. Otherwise, for q_j's subtree resources that were not reachable to p_t, p_t becomes reachable and the RP[] matrix is updated to reflect the change in line 38.

As a summary, the Release_Update_Reachability kernel updates all sink nodes in q_j's sub-tree to p_t. The process p_i that released the resource is no longer reachable

from q_j and its sub-tree, so the RP[] matrix is updated accordingly. The final steps in our computation require that all p_t's sub-tree resources that were previously able to reach q_j be removed from the RR[] matrix. Conversely, if q_j's sub-tree resources were not reachable to p_t, p_t now becomes reachable and the RP[] matrix is updated.

3.6 Supplementary Kernels

One may notice that there were three additional kernels in the overview code that were not discussed so far, the BitMatrix_Tranpose, Tile_Transpose, and Row_Reduction kernels. The combination of the BitMatrix_Transpose and Tile_Transpose kernels enables us to transpose our bit-vector matrices, which ensures that coalesced global memory accesses occur in our Row_Reduction kernels. This is why the BitMatrix_Transpose and Tile_Transpose kernels always precede the Row_Reduction kernel.

While these kernels are supplementary to the GPU-OSDDA functionality, they provide substantial speedups with regard to the run-time of our algorithm. Without performing the bit-matrix transpose that advocates global memory coalescing in kernels, we would have seen a great loss in efficiency (since memory coalescing would not occur) and run-time.

3.6.1 Bit-vector Matrix Transpose Kernels

Performing the transpose of a bit-vector matrix can be a complicated task. Fortunately, this computation has been studied by [13]; however, the transpose in [13] only works on a 32-bit $\times$ 32-bit matrix. Therefore, to make this solution fit to our problem (as our matrices are much larger) and to parallelize the computation, we sub-divided the transpose of our matrices into 32-bit $\times$ 32-bit tiles (seen as TILE_DIM in Algorithm 1). Then we launch TILE_DIM blocks and TILE_DIM threads in the BitMatrix_Transpose kernel. This ensures in the first step that each thread transposes a 32-bit $\times$ 32-bit tile of the matrix, as shown in Figure 6.

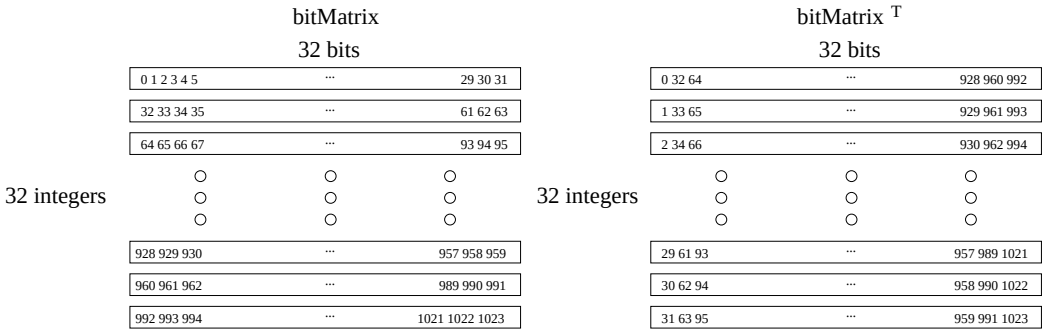


Figure 6. Inner tile bit-matrix transpose.

After the elements within each tile have been transposed, the Tile_Transpose kernel performs an outer tile transpose, i.e., tile by tile as shown in Figure 7, to place all elements in the correct positions. By performing the transpose of our entire matrix in tiles, we were not only able to parallelize the transpose but also able to enable coalesced global memory accesses for its following kernels, thereby leading to a fast bit-vector matrix transpose operation.

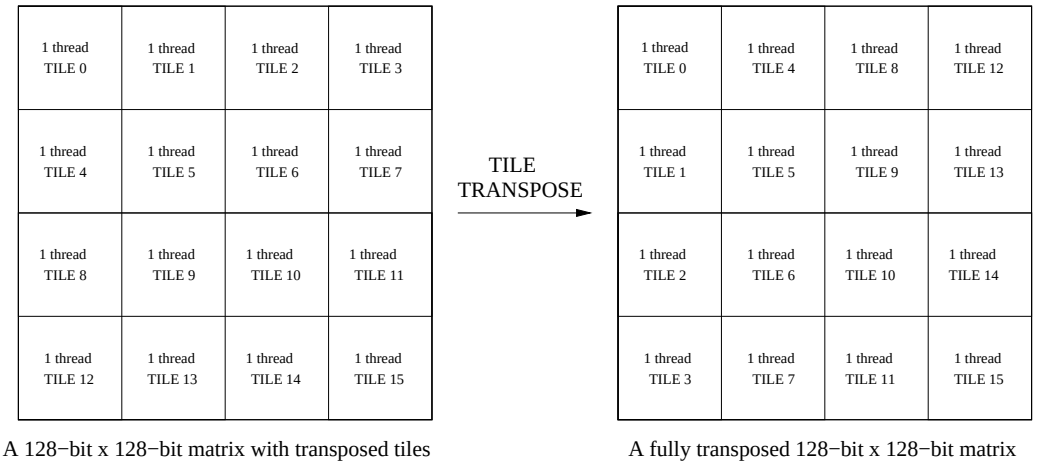


Figure 7. Outer tile bit-matrix transpose.

3.6.2 Bit-vector Row Reduction Kernel

Our Row_Reduction kernel also greatly benefits from our bit-vector approach. All of the reductions in GPU-OSDDA are used to check to see if the row or column of a particular matrix is zero. This works well with the bit-vector approach. When we perform our reduction, we simply add each integer of a particular row or column together (an add reduction) and determine if the total is zero or not. This allowed us to compute the row or column reduction 32X faster than if we had not taken the bit-vector approach in storing our adjacency matrices. To optimize our reduction kernels even further, we perform the first add of the reduction when we populate shared memory, thus allowing us to launch half the number of threads required in a standard reduction. From that point, we perform the reduction while unrolling the last warp utilizing the *warpReduce* function found in [14].

4. Experimentation and Results

All experiments were performed on an Intel®Core i7 CPU @ 2.8 GHz with 12 GB RAM. The CUDA GPU-OSDDA implementation was tested with different GPUs: Tesla C2050, Tesla K20c, and Titan X. The Tesla C2050 has 14 SMs (448 CUDA Cores) with 3 GB Global Memory, the Tesla K20c has 13 SMXs (2496 CUDA Cores) with 5 GB Global Memory, and the Titan X has 24 SMs (3072 CUDA Cores) with 12 GB Global Memory.

A serial version of GPU-OSDDA is implemented using the C language, referred to as CPU-OSDDA. We attempted to create a multi-threaded version in CPU using OpenMP. However, there was no speedup because the algorithm is not computationally intensive but is bounded by memory read and write. In this case, the overhead of thread management makes the run-time slower. Therefore, we use the serial version CPU-OSDDA to compare the performance with parallel computation.

To verify the correctness of our algorithm, both CPU-OSDDA and GPU-OSDDA were tested using RAGs of different sizes and complexities. To create the RAGs, we prepare a list of events. CPU will send out each event sequentially and launch GPU kernels depending on the event type.

The response time for each type of events is measured. The Resource Request Granted Event has trivial response time due to its simple algorithm. For Resource Blocked Event, we created an event list of a worst-case scenario $p_1 \rightarrow q_1 \rightarrow p_2 \rightarrow q_2 \ldots \rightarrow p_M \rightarrow q_M \rightarrow p_1$ ($2M$ events in total, where M is the number of processes).

Then the time to update all the state matrices for the last blocked event $p_M \rightarrow q_M$ before deadlock happens is recorded in Table 2. Using NVIDIA Visual Profiler [15], we measured the communication time and the computation time of the GPUs. As shown in Figure 9, the larger the input size is, the longer the computation time it takes out of total time. As the communication overhead has less effect, we were able to gain higher speedup for larger input sizes.

In a similar manner, for Resource Release Event, we tested the average-case scenario by modifying the length of resource-process chain which has the same sink to the process releasing the resource. The reason to test for average-case is to provide a big picture on the average speedup of using GPU under typical scenarios. The time is recorded in Table 3, which shows less speedup than Resource Blocked Event because in CPU-OSDDA, the algorithm itself for Resource Release Event has linear complexity, whereas Resource Blocked Event has polynomial complexity.

Figures 8 and 10 depict the associated speedups of each set size on each piece of target hardware. From the result, the increasing size of RAG dramatically increases CPU-OSDDA's run-time. However, GPU-OSDDA scales well with increasing process and resource amounts.

Table 2. Response time to blocked event (worst-case) (ms/speedup)

Input	CPU-OSDDA	Tesla C2050	Tesla K20c	Titan X
512×512	0.819 / 1X	0.130 / 6.2X	0.075 / 11.0X	0.134 / 6.1X
1024×1024	5.251 / 1X	0.136 / 38.5X	0.081 / 65.2X	0.137 / 38.4X
2048×2048	14.64 / 1X	0.189 / 77.3X	0.109 / 134.1X	0.144 / 101.5X
4096×4096	62.68 / 1X	0.302 / 207.8X	0.211 / 296.9X	0.192 / 326.9X
8192×8192	253.58 / 1X	0.830 / 305.6X	0.547 / 463.7X	0.426 / 595.0X

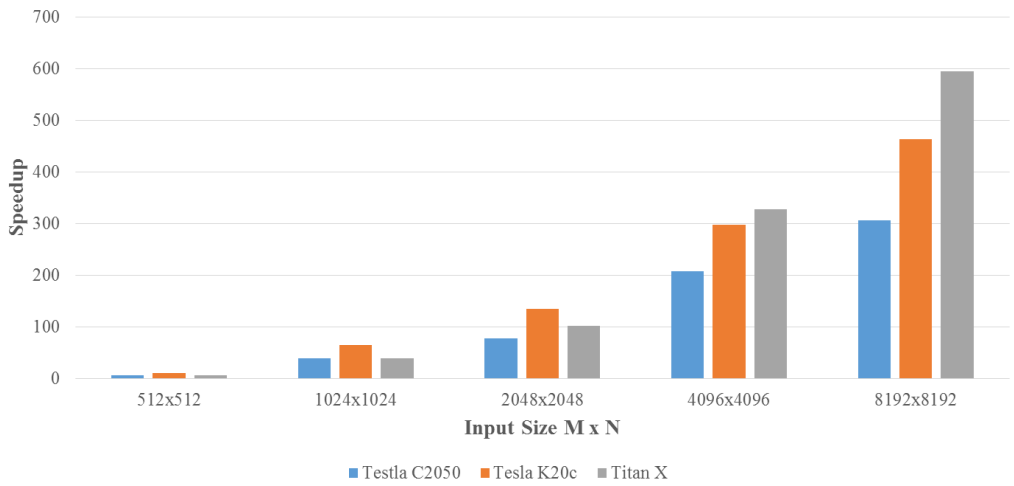


Figure 8. GPU-OSDDA speedup for blocked event (worst-case) against CPU version

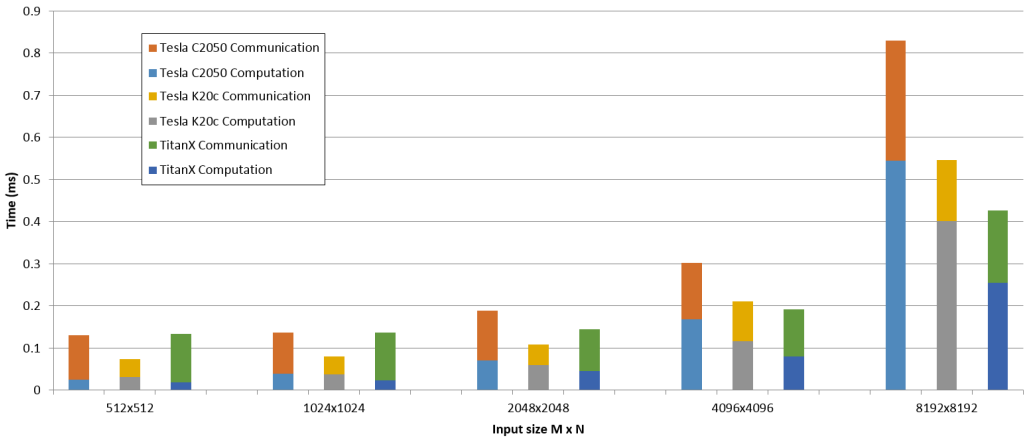


Figure 9. Computation and Communication time for the GPUs in GPU-OSDDA

Table 3. Response time to resource released event (average-case) (ms/speedup)

Input	CPU-OSDDA	Tesla C2050	Tesla K20c	Titan X
512×512	0.016 / 1X	0.058 / 0.27X	0.050 / 0.31X	0.070 / 0.22X
1024×1024	0.046 / 1X	0.061 / 0.75X	0.052 / 0.884X	0.066 / 0.70X
2048×2048	0.240 / 1X	0.070 / 3.44X	0.060 / 3.96X	0.075 / 3.20X
4096×4096	0.921 / 1X	0.106 / 8.67X	0.067 / 13.79X	0.085 / 10.8X
8192×8192	3.334 / 1X	0.214 / 15.6X	0.145 / 23.0X	0.133 / 25.1X

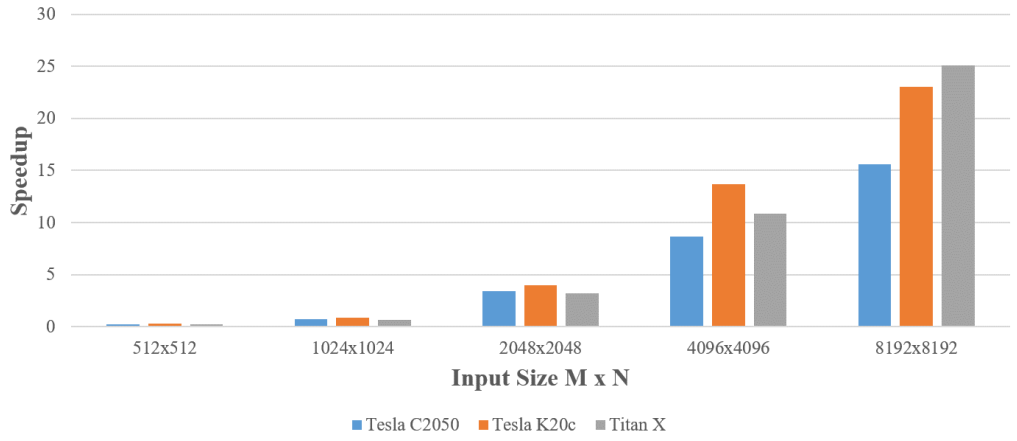


Figure 10. GPU-OSDDA speedup for released event (average-case) against CPU version

5. Conclusion

A new approach to deadlock detection for single-unit systems on GPU has been devised and developed using CUDA C. By leveraging facts about single-unit systems, we were able to devise a bit-vector technique for storing our matrices, which led to efficient algorithmic computations and drastically saved memory space on the GPU. These factors allow GPU-OSDDA to handle systems with increasing

amounts of processes and resources. GPU-OSDDA is the first of any deadlock detection algorithm able to handle such a high number of processes and resources while being able to provide practical run-times. In the past, this has been limited by either lack of parallelism, size constraints, or a combination of both. Since GPU-OSDDA performs deadlock computation/detection on the GPU, our algorithm acts as an interactive service to resource events occurring on the CPU. Our experimental results show promising speedups, thus making GPU-OSDDA a viable solution to deadlock detection on single-unit resource systems with a large number of processes and resources.

Future work would include further optimizing GPU processing time using dynamic parallelism. In addition, the performance of GPU-OSDDA under a real-time operating system should be investigated.

References

[1] *Deadlock* (2014), http://en.wikipedia.org/wiki/Deadlock.

[2] R.C. Holt, *Some deadlock properties of computer systems*, ACM Comput. Surv. 4 (1972), pp. 179–196.

[3] J.G. Kim, *An algorithmic approach on deadlock detection for enhanced parallelism in multiprocessing systems*, in *International Symposium on Parallel Algorithms/Architecture Synthesis*, 1997, pp. 233–238.

[4] J.K. Kim and K. Koh, *A O(1) Time Deadlock Detection Scheme in a Single Unit and Single Request Multiprocessor System*, in *IEEE International Conference on EC3-Energy, Computer, Communication and Control Systems*, Vol. 2, August, (1991), pp. 219–223.

[5] A. Shoshani and E. Coffman, *Detection, prevention and recovery from deadlocks in multiprocess multiple resource systems*, Princeton University, (1969).

[6] T.F. Leibfried, *A deadlock detection and recovery algorithm using the formalism of a directed graph matrix*, SIGOPS Operating Systems Review 23 (1989), pp. 45–55.

[7] P.H. Shiu, Y. Tan, and V.J. Mooney, *A novel parallel deadlock detection algorithm and architecture*, in *International Symposium on Hardware/Software Codesign*, (2001), pp. 73–78.

[8] X. Xiao and J.J. Lee, *A novel parallel deadlock detection algorithm and hardware for multiprocessor system-on-a-chip*, IEEE Computer Architecture Letters 6 (Feb. 2007), pp. 41–44.

[9] X. Xiao and J.J. Lee, *A true O(1) parallel deadlock detection algorithm for single-unit resource systems and its hardware implementation*, IEEE Transactions on Parallel and Distributed Systems 21 (Jan. 2009), pp. 4–19.

[10] X. Xiao and J.J. Lee, *A novel O(1) deadlock detection methodology for multiunit resource systems and its hardware implementation for system-on-chip*, IEEE Transactions on Parallel and Distributed Systems 19 (Dec. 2008), pp. 1657–1670.

[11] X. Xiao and J.J. Lee, *A parallel multi-unit resource deadlock detection algorithm with O(log2(min(m,n))) overall run-time complexity*, Elsevier Journal of Parallel and Distributed Computing, 71 (2011), pp. 938–954.

[12] M. Maekawa, A.E. Oldehoeft, and R.R. Oldehoeft, Operating Systems: Advanced Concepts, Benjamin/Cummings Pub. Co., Menlo Park, CA, 1987.

[13] H.S. Warren, *Hacker's Delight*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[14] M. Harris, *et al.*, *Optimizing parallel reduction in CUDA*, NVIDIA Developer Technology 2 (2007).

[15] *NVIDIA Visual Profiler* (2015), http://developer.nvidia.com/nvidia-visual-profiler.